Project Report
Stock Market Simulator

Table of Contents

Created by:
Ralph Khairallah

Date: June 5, 2025

1 Introduction

As part of a personal project, I developed an application called **Stock Market Simulator**. The main goal of this project is to provide an educational tool that allows any type of user—whether a beginner or more experienced—to learn how to invest virtually in the stock market.
Through a simple yet complete interface, users can buy and sell fictitious stocks, monitor the performance of their portfolio, and observe in real time the impact of their decisions. This simulator aims to make key concepts of the investment world more accessible: risk management, performance tracking, and strategic decision-making across various market scenarios such as financial crashes, periods of high inflation, or exceptional events like the COVID-19 crisis.
This report outlines the various stages of the application's design and development. It details the chosen architecture, the technical decisions made over time, the main implemented features, as well as the challenges encountered and the solutions applied.

2 Main Features

The main user stories developed in this application are as follows:

Importing Data

The goal of this user story is to efficiently import financial product data over extended periods. For instance, the application must access opening and closing prices (prices at the start and end of the day) of a specific stock over a given timeframe.

This functionality was prioritized early in the development and was completed.

Storing Data

Once the data has been retrieved, it must be stored efficiently to allow easy access and reuse later.

This feature was implemented progressively. While some aspects could still be optimized, the current storage system is functional and allows the application to use the data effectively.

Displaying a Financial Product's Price

To buy a financial product, the user must be able to track its price evolution over time. The objective here is to provide a dynamic display of price curves, with various buttons to show or hide specific financial indicators.

The main display was implemented during the early development phases and later enriched with interactive buttons.

Buying and Selling a Financial Product

The user must be able to buy or sell any financial product. The portfolio balance is automatically updated based on the purchase or sale price, and assets are added to or removed from the user's holdings.

This feature was fully developed afterward.

Accessing Any Financial Product

The application includes a search bar that allows the user to access any financial product. This gives users great flexibility, as they are not limited to a predefined list.

This feature is operational, although some improvements could still be made.

Creating an Account and Resuming a Session

Users can create an account to save their progress and resume it later. This requires storing user data, such as usernames and passwords.

The foundations of this feature are in place: sign-up and login interfaces are functional, and credentials are stored. However, the full authentication system still needs to be secured.

Advancing Through Time

Instead of tracking the market in real time, the application simulates an accelerated timeline. When a user selects a stock, its price is displayed over a predefined timeframe (e.g., January to March 2025). The user can then advance through time to instantly see how the stock evolves.

This system is implemented, but users cannot yet choose the initial date freely: data always starts from early 2025.

Jumping to a Historical Event

To support its educational goal, the application aims to allow users to jump directly to a time period marked by a major historical event, in order to observe its impact on the markets.

This feature has not yet been developed.

3 Architecture

3.1 Graphical Interface

3.1.1 Class Diagram

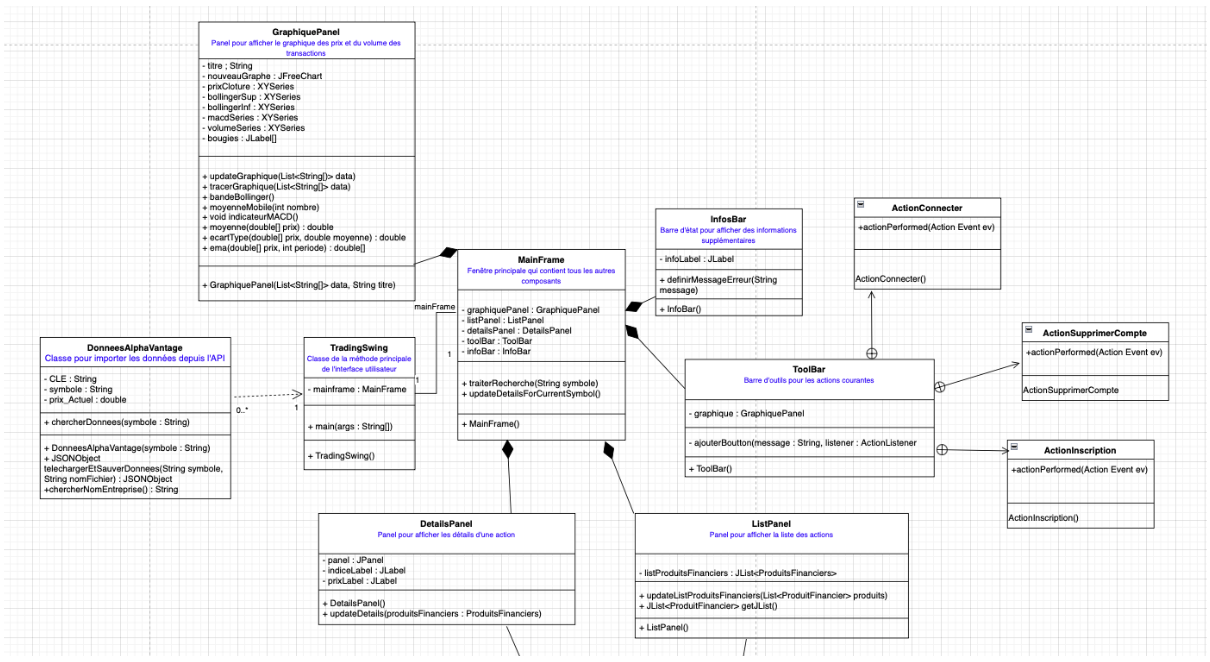
GraphiquePanel
Panel pour afficher le graphique des prix et du volume des transactions
- titre ; String
- nouveauGraphe : JFreeChart
- prixCloture : XYSeries
- bollingerSup : XYSeries
- bollingerInf : XYSeries
- macdSeries : XYSeries
- volumeSeries : XYSeries
- bougies : JLabel[]
+ updateGraphique(List<String[]> data)
+ tracerGraphique(List<String[]> data)
+ bandeBollinger()
+ moyenneMobile(int nombre)
+ void indicateurMACD()
+ moyenne(double[] prix) : double
+ ecartType(double[] prix, double moyenne) : double
+ ema(double[] prix, int periode) : double[]
+ GraphiquePanel(List<String[]> data, String titre)
MainFrame
Fenêtre principale qui contient tous les autres composants
- graphiquePanel : GraphiquePanel
- listPanel : ListPanel
- detailsPanel : DetailsPanel
- toolBar : ToolBar
- infoBar : InfoBar
+ traiterRecherche(String symbole)
+ updateDetailsForCurrentSymbol()
+ MainFrame()
mainFrame
1
1
DonneesAlphaVantage
Classe pour importer les données depuis l'API
- CLE : String
- symbole : String
- prix_Actuel : double
+ chercherDonnees(symbole : String)
+ DonneesAlphaVantage(symbole : String)
+ JSONObject telechargerEtSauverDonnees(String symbole, String nomFichier) : JSONObject
+chercherNomEntreprise() : String
0..*
1
TradingSwing
Classe de la méthode principale de l'interface utilisateur
- mainframe : MainFrame
+ main(args : String[])
+ TradingSwing()
InfosBar
Barre d'état pour afficher des informations supplémentaires
- infoLabel : JLabel
+ definirMessageErreur(String message)
+ InfoBar()
ActionConnecter
+actionPerformed(Action Event ev)
ActionConnecter()
ActionSupprimerCompte
+actionPerformed(Action Event ev)
ActionSupprimerCompte
ToolBar
Barre d'outils pour les actions courantes
- graphique : GraphiquePanel
- ajouterBoutton(message : String, listener : ActionListener
+ ToolBar()
ActionInscription
+actionPerformed(Action Event ev)
ActionInscription()
DetailsPanel
Panel pour afficher les détails d'une action
- panel : JPanel
- indiceLabel : JLabel
- prixLabel : JLabel
+ DetailsPanel()
+ updateDetails(produitsFinanciers : ProduitsFinanciers)
ListPanel
Panel pour afficher la liste des actions
- listProduitsFinanciers : JList<ProduitsFinanciers>
+ updateListProduitsFinanciers(List<ProduitFinancier> produits)
+ JList<ProduitFinancier> getJList()
+ ListPanel()

Figure 1: Part of the UML diagram for the graphical interface

3.1.2 Class Explanations

➢ **GraphiquePanel** is a `JPanel` that integrates two charts generated with JFreeChart:

- the main chart displays the closing price evolution over time;
- a secondary chart shows the trading volumes.

When `setData(List<String[]> data)` is called, the chart updates with the new stock data. The viewing window can be expanded with the "Advance" button.

The class also computes the **visible price on the chart** (via `prixGraphe`) based on the current displayed date, which is essential if a user wants to buy a stock at the chart price.

`GraphiquePanel` uses an `Observer` mechanism with `PropertyChangeSupport` to notify other components (like `MainFrame`) when the chart price changes.

➢ **DonneAlphaVantage** is a class used to interact with the `Alpha Vantage` web API. It is responsible for:

- `chercherDonne()`: retrieves daily data (open, high, low, close, volume) either via the API or from a local JSON cache file;

  - `chercherNomEntreprise()`: queries the OVERVIEW API to retrieve the full company name from its symbol.

  The data is cached locally to reduce API calls (limited per day). On each search, the app checks if the local file is still valid by date; otherwise, it downloads new data.

- ➢ **MainFrame** is the main class of the graphical interface. It extends `JFrame` and aggregates all the components (toolbar, chart, left panel, right panel, info bar, etc.).

  It coordinates user interactions and updates components accordingly. Its responsibilities include:

  - `traiterRecherche(String symbole)`: triggered when a stock is searched. It:
    - ∗ uses `DonneAlphaVantage` to load the data;
    - ∗ updates the chart via `graphiquePanel.setData(...)` ;
    - ∗ extracts the chart price (adjusted in time) and updates `detailsPanel`;
    - ∗ adds the stock to the left list if it is not already there.
  - `updateDetailsForCurrentSymbol()`: listens for chart changes and dynamically updates stock details.

- ➢ **TradingSwing** is the class containing the **main method**. It initializes the UI by creating an instance of `MainFrame`, the graphical entry point of the application. It uses `SwingUtilities.invokeLater` to follow Swing best practices (executing code on the GUI thread).

- ➢ **ToolBar** is a graphical toolbar that includes all action buttons available in the app (search, indicators, login, signup, etc.). It extends `JToolBar` and appears at the top of the window.

  It includes:

  - a text field `champRecherche` where the user can enter a stock symbol;
  - a `Search` button that triggers the `traiterRecherche()` method in `MainFrame` to display stock data;
  - buttons to display chart indicators such as Bollinger bands or moving averages;
  - signup and login buttons with their respective interfaces.

  The search field also responds to the "Enter" key via an `ActionListener`. Errors (invalid symbols) or successes (chart display) are communicated via the `InfoBar`.

- ➢ **InfoBar** is a simple text bar located at the bottom of the window. It extends `JLabel` and displays:

  - informative messages (e.g., “Data for MCD”) via `mettreAJourInfo()`;
  - error messages with special styling (yellow background, red text, border) via `definirMessageErreur()`.

  It provides immediate user feedback after each interaction.

➢ **ListPanel** represents the left column of the user interface. It is organized using a `BorderLayout` and includes three essential components:

- a `JList<ProduitFinancier>` at the top, displaying the financial products the user has searched for;
- the user's portfolio view (`VuePortefeuille`) in the center;
- the portfolio controller (`ControlerPortefeuille`) at the bottom, for buying or selling products.

This structure clearly follows the **MVC** architecture:

- the model is represented by `Portefeuille`;
- the view is `VuePortefeuille`;
- the controller is `ControlerPortefeuille`.

The method `updateListProduitsFinanciers(...)` reloads the entire visible stock list. When a searched product is not yet listed, it is added to this component.

➢ **DetailsPanel** is a graphical panel located on the right side of the user interface. It provides a **visual summary** of the key data related to a selected financial product:

- the stock **symbol** (e.g., "AAPL", "TSLA");
- the current **price** of the day;
- the **price displayed on the chart**, if time-adjusted;
- the **percentage variation** between today and the previous day.

All these elements are updated via the `updateDetails()` method when a product is selected from the list or searched for.

`DetailsPanel` allows the user to get an instant and clear overview of the stock's current state without needing to analyze the chart.

3.2 User and Financial Management Components

3.2.1 Class Diagram

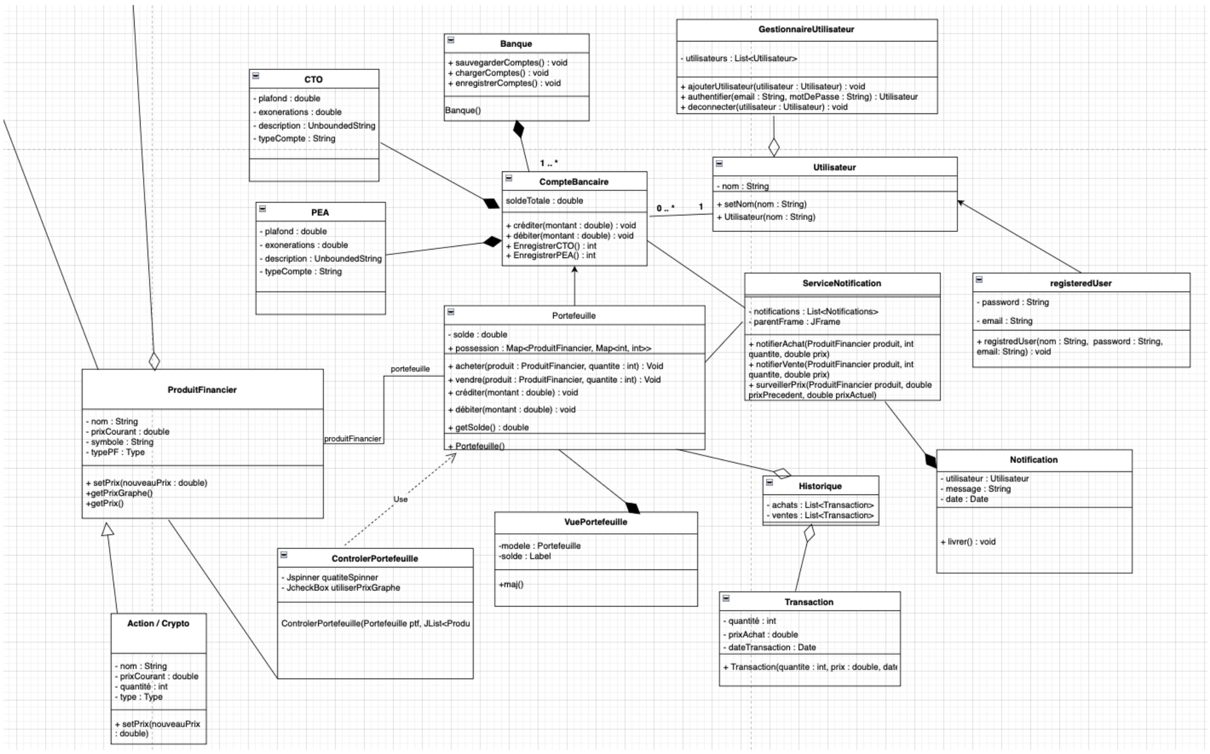
Banque
+ sauvegarderComptes() : void
+ chargerComptes() : void
+ enregistrerComptes() : void
Banque()
GestionnaireUtilisateur
- utilisateurs : List<Utilisateur>
+ ajouterUtilisateur(utilisateur : Utilisateur) : void
+ authentifier(email : String, motDePasse : String) : Utilisateur
+ deconnecter(utilisateur : Utilisateur) : void
CTO
- plafond : double
- exonerations : double
- description : UnboundedString
- typeCompte : String
PEA
- plafond : double
- exonerations : double
- description : UnboundedString
- typeCompte : String
1 .. *
CompteBancaire
soldeTotale : double
+ créditer(montant : double) : void
+ débiter(montant : double) : void
+ EnregistrerCTO() : int
+ EnregistrerPEA() : int
0 .. *
1
Utilisateur
- nom : String
+ setNom(nom : String)
+ Utilisateur(nom : String)
ServiceNotification
- notifications : List<Notifications>
- parentFrame : JFrame
+ notifierAchat(ProduitFinancier produit, int quantite, double prix)
+ notifierVente(ProduitFinancier produit, int quantite, double prix)
+ surveillerPrix(ProduitFinancier produit, double prixPrecedent, double prixActuel)
registeredUser
- password : String
- email : String
+ registredUser(nom : String, password : String, email: String) : void
Portefeuille
- solde : double
+ possession : Map<ProduitFinancier, Map<int, int>>
+ acheter(produit : ProduitFinancier, quantite : int) : Void
+ vendre(produit : ProduitFinancier, quantite : int) : Void
+ créditer(montant : double) : void
+ débiter(montant : double) : void
+ getSolde() : double
+ Portefeuille()
portefeuille
produitFinancier
ProduitFinancier
- nom : String
- prixCourant : double
- symbole : String
- typePF : Type
+ setPrix(nouveauPrix : double)
+getPrixGraphe()
+getPrix()
Use
Notification
- utilisateur : Utilisateur
- message : String
- date : Date
+ livrer() : void
Historique
- achats : List<Transaction>
- ventes : List<Transaction>
VuePortefeuille
-modele : Portefeuille
-solde : Label
+maj()
ControlerPortefeuille
- Jspinner quatiteSpinner
- JcheckBox utiliserPrixGraphe
ControlerPortefeuille(Portefeuille ptf, JList<Produ
Transaction
- quantité : int
- prixAchat : double
- dateTransaction : Date
+ Transaction(quantite : int, prix : double, date
Action / Crypto
- nom : String
- prixCourant : double
- quantité : int
- type : Type
+ setPrix(nouveauPrix : double)

Figure 2: Part of the UML diagram for user and financial management components

3.2.2 Class Explanations

➢ **UserStorage** is a class responsible for **in-memory user management**. It maintains two lists: a general list `users` (all users) and a list `registredUsers` (users with email/password). It provides methods to:

- add a user;
- remove a user (by username or email);
- retrieve stored users;
- display users.

This class helps centralize registration and local account management before persistence.

➢ **User** is the base class representing an application user. It has a `username` and a `CompteBancaire`. The user interacts with portfolios through this account. It defines the minimal identity of a user.

➢ **RegistredUser** inherits from `User` and adds authentication data: `email` and `password`. It is used for login and signup, allowing the distinction between authenticated and non-authenticated users.

➢ **Notification** represents a user-facing message, with a date and type (buy, sell, alert, etc.). Each notification has a `message`, a `type`, and a `timestamp`. It is a simple structure for tracking meaningful events.

➢ **ServiceNotification** centralizes **user notification management**. It stores a list of notifications and provides:

- creation and display of messages for each **buy or sell** action;
- sending of **price alert notifications** in case of significant variation;
- graphical display of alerts using `JOptionPane`.

It works closely with `Portefeuille` and `MainFrame`.

➢ **CompteBancaire** models a user account containing several portfolios (`Portefeuille`, `CTO`, or `PEA`). It allows:

- **crediting** or **debiting** the total balance;
- **registering portfolios**, automatically assigning them a key;
- managing the user's holdings.

It is the interface between financial products and user actions.

➢ **Transaction** describes a buy/sell operation performed by the user. It includes:

- the `quantity` of units exchanged;
- the `unit purchase price`;
- the `transaction date`.

It is used by `Portefeuille` to track purchase history.

➢ **GestionnaireUtilisateur** is a utility class responsible for **user persistence**. It uses the `Gson` library to save/load users in a JSON file (`utilisateur.json`). It also provides:

- a method to log in using email and password;
- a method to save a user after registration.

It acts as a bridge between local storage and the application's runtime state.

➢ **CTO** and **PEA** represent two types of investment accounts, each with its own tax ceilings and legal definitions. Both inherit from `CompteBancaire`.

➢ **Banque** handles the creation and initialization of these different accounts.

➢ Within a `CompteBancaire`, the **Portefeuille** manages financial product holdings and transactions. It maintains the current state (quantity of each asset) and provides methods to buy, sell, credit, and debit.

➢ **Portefeuille** is the core model for a user. It handles:

- the available **cash balance** for transactions;

- the **current asset holdings** (quantity, purchase price, etc.);
- the list of **completed transactions**.

It uses a map `Map<ProduitFinancier, List<Transaction»` to track purchased quantities and associated prices. Each key is a product (stock, crypto, etc.), and the value is the list of `Transaction` objects for that product.

It provides several key methods:

- `acheter(ProduitFinancier, int, double)`: adds a purchase transaction after balance verification;
- `vendre(ProduitFinancier, int)`: removes a quantity if available and credits the balance;
- `crediter()` and `debiter()`: manipulate the balance directly;
- `getQuantite(...)`: returns the total number of units held for a product;
- `getPossessions()`: accesses the holdings map.

The class inherits from `Observable`, allowing `VuePortefeuille` to subscribe to it. Any change (buy, sell, credit, debit) triggers an automatic update of the graphical view.

When buying or selling, `Portefeuille` also sends a notification through `ServiceNotification`, especially for price change alerts.

➢ **ProduitFinancier** is an `abstract class` representing any asset that can be held in a portfolio: a stock, cryptocurrency, ETF, etc.

It defines shared properties and behaviors for all products:

- `getNom()` and `getSymbole()`: return the asset name and ticker;
- `getPrixCourant()`: returns the current asset price;
- `getPrixGraphe()`: returns the price at the chart's current position;
- `setPrixCourant(...)` and `setPrixGraphe(...)`: update the prices dynamically.

This class is meant to be extended. For instance:

- `ActionGenerique` is a concrete subclass for dynamically searched stocks;
- `CryptoMonnaie` or other products can be added later using the same structure.

This abstraction allows the application to manage all types of assets uniformly — for display, trading, and graphical analysis.

➢ **VuePortefeuille**

The `VuePortefeuille` class is the **graphical view of the user's portfolio**. It extends `JPanel` and displays two main elements:

- a `JLabel` showing the user's **current balance**;
- a `JTable` listing the owned financial products (name, quantity, purchase price).

`VuePortefeuille` observes the `Portefeuille` model (which inherits from `Observable`). When a transaction occurs (buy or sell), the model automatically notifies the view, which calls `maj()` to update the display.

`maj()` iterates over the portfolio's holdings map and reloads the table. Each row shows:

- the product name (`ProduitFinancier.getNom()`);
- the quantity held (extracted from `Transaction`);
- the purchase price.

`VuePortefeuille` thus provides a consistent and synchronized interface with the model data.

➢ **ControlerPortefeuille**

The `ControlerPortefeuille` class is a graphical component acting as the **controller for portfolio interaction**. It enables the user to buy and sell selected financial products.

It includes:

- a `JSpinner` to choose the quantity to buy or sell;
- a `JCheckBox` to select the use of the **chart price** instead of the current market price;
- two `JButtons`: "Buy" and "Sell".

When a button is clicked:

1. the selected product from the `JList<ProduitFinancier>` is retrieved;
2. the quantity is read from the `JSpinner`;
3. the price is determined based on the `JCheckBox` state (`getPrixGraphe()` or `getPrixCourant()`);
4. the method `ptf.acheter(...)` or `ptf.vendre(...)` is called;
5. if the transaction is successful, the portfolio view is automatically refreshed.

Dialog boxes are shown in case of error (no selected product, insufficient funds, or invalid quantity).

`ControlerPortefeuille` and `VuePortefeuille` are both embedded within `ListPanel`, ensuring clear separation between business logic (model), interactions (controller), and display (view).

4 System Dynamics

4.1 Object Collaboration

When a user wishes to buy a financial product, the interaction begins through the user interface, via **ControlerPortefeuille**, where the user clicks a dedicated button. This event is captured by the **MainFrame**, which acts as the entry and coordination point of the application.

The **MainFrame** then delegates the purchase operation to the user's portfolio, represented by the **Portefeuille** class. The latter consults the **CompteBancaire** to ensure sufficient funds are available, and debits the required amount.

The portfolio then updates the financial product holdings and creates a **Transaction** instance to record the purchase. This transaction is added to the user's **History**. Meanwhile, a notification is generated via **ServiceNotification** and sent to the user to confirm the successful operation.

Finally, the **MainFrame**, through the **InfoBar** object, displays a visual confirmation message.

4.2 Data Flow in the System

Data flows through the simulator in a structured manner. For example, stock prices are fetched at startup or when pressing the "Advance" button on the **ToolBar**, via the **DonnéesAlphaVantage** class, which connects to the external **AlphaVantage** API.

This data is then stored in **ProduitFinancier** objects, and displayed through interfaces such as **GraphiquePanel** or **ListPanel**, either as charts or lists.

When a user makes a purchase, the inputs from the interface become transactions and are used to update the state of the portfolio and bank account. They are also forwarded to the notification service to inform the user and to display components like **InfoBar** for real-time updates.

Additionally, market indicators such as moving averages and Bollinger bands are recalculated using price data stored in the time series (XYSeries) of the **GraphiquePanel**, ensuring dynamic market visualization.

4.3 Execution Dependencies

The graphical interface, notably the **DetailsPanel** and **ListPanel** classes, depends on business components such as **Portefeuille**, **CompteBancaire**, and **ServiceNotification**. These are called upon to execute typical operations like buying or selling.

Business logic modules, in turn, rely on market data provided by **DonnéesAlphaVantage**, which is an external dependency tied to the API.

Finally, the notification and display mechanisms rely on coordination between core logic and the interface, through calls to **ServiceNotification** and updates to the **InfoBar**.

5 Main Design and Implementation Choices

5.1 Data Import and Storage

To develop this application, access to a large amount of financial product data was essential, especially since users needed to be able to access any product of their choice. For this purpose, I used an API to fetch the data of a stock in JSON format using a key and the stock symbol. A method was developed to process this data and convert it into a usable table, allowing the application to handle it efficiently. Users can simply enter the name of a stock in the search bar to view its price or purchase a specific quantity.

In the initial version, data was not directly saved to a JSON file after being imported. Each time the application was restarted or a new chart was displayed, a new API request was made.

This approach had two main drawbacks:

- The application was slowed down due to API response times.
- The 20-request API quota was quickly reached, preventing further actions.

To address these issues, I chose to store the data locally in a JSON file upon import.

Thus, for every attempt to import a new stock, the application first checks if a corresponding local JSON file already exists, in order to avoid unnecessary API requests.

5.2 Dynamic Data Update and User Interaction

A key feature of this application is its ability to dynamically respond to user actions. Through the `ToolBar`, the user can search for a stock by entering its symbol and pressing "Enter" or clicking the `Search` button, triggering the retrieval or loading of data.

Each searched stock is added to an interactive list on the left side of the screen. Clicking on a stock automatically updates the central chart, as well as the information panel on the right (symbol, current price, chart price, variation).

Furthermore, when purchasing a product, the user can choose between the current market price or the chart-displayed price at a specific moment (simulated via the `Advance` button). This level of precision enhances the realism of the simulator by allowing users to buy stocks at a specific moment in the simulated timeline.

6 Development Organization

At the start of development, I took time to define the project's objectives and design the UML architecture. I structured the project into two main parts: the graphical user interface (GUI) and the business logic core, including user and portfolio management.

To organize tasks efficiently, I used Trello. This allowed me to break development into clear user stories, helping track progress and prioritize features.

Although some parts were developed remotely, I found that working in person, in the school's computer rooms, side by side with other students (even those not involved in this project), helped me maintain productivity. These sessions allowed me to focus, code more efficiently, and test class interactions in a conducive environment.

This organization allowed me to build a complete application with a modular structure that supports future extensions.